PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F | A2

(11) International Publication Number: WO 98/09208

(43) International Publication Date: 5 March 1998 (05.03.98)

(21) International Application Number: PCT/US97/14842

(22) International Filing Date: 21 August 1997 (21.08.97)

(30) Priority Data:
08/705,881 28 August 1996 (28.08.96) US

(71) Applicant: WIND RIVER SYSTEMS, INC. [US/US]; 1010 Atlantic Avenue, Alameda, CA 94501 (US).

(72) Inventors: FOGELIN, John, C.; 250 Coleridge Street, San Francisco, CA 94110 (US). SMITH, Colin; 22, impasse de la Capitainerie, F-56860 Sene (FR). WILNER, David, N.; 2733 Claremont Boulevard, Berkeley, CA 94705 (US). HARTMAN, John; 4680 Davenport Avenue, Oakland, CA 94619 (US). LONG, Kent, D.; 6109 McBride Avenue, Richmond, CA 94805 (US). STEIMAN-CAMERON, Debbie; 30 Waterside Circle, Redwood City, CA 94065 (US). SHEPARD, Marc; 4584 Santa Rita Road, Richmond, CA 94803 (US). ZHOU, Yiwen; 2020 Franciscan Way #208, Alameda, CA 94501 (US). WADDINGTON, Simon; 3 Embarcadero West #341, Oakland, CA 94607 (US). MAISONNEUVE, Philippe; 1, allée Louis Aubert, F-56000 Vannes (FR). DARLET, Pierre-Alain; 17, square les Bouleaux, F-56610 Arradon (FR). PREYSSLER, Thierry; 2615 Telegraph Avenue #203, Berkeley, CA 94704 (US). ORIOT, Jean-Claude; 3, rue de Ranquin, F-56860 Sene (FR).

(74) Agents: MENDENHALL, Larry et al.; Townsend and Townsend and Crew LLP, 8th floor, Two Embarcadero Center, San Francisco, CA 94111 (US).

(81) Designated States: JP, European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
Without international search report and to be republished upon receipt of that report.

(54) Title: A TOOL FOR SOFTWARE DIAGNOSIS DIVIDED ACROSS MULTIPLE PROCESSORS

(57) Abstract

A target/host computer system wherein traffic on the connecting communications link and the size of the agent in the target are reduced by using Gophers, debugging services on demand, shared run-time libraries, quantized streaming differential caches, host-based target memory management, virtual I/O and in-circuit emulation.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece			TR	Turkey
BG	Bulgaria	HU	Hungary	ML	Mali	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MN	Mongolia	UA	Ukraine
BR	Brazil	IL	Israel	MR	Mauritania	UG	Uganda
BY	Belarus	IS	Iceland	MW	Malawi	US	United States of America
CA	Canada	IT	Italy	MX	Mexico	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NE	Niger	VN	Viet Nam
CG	Congo	KE	Kenya	NL	Netherlands	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NO	Norway	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	NZ	New Zealand		
CM	Cameroon			PL	Poland		
CN	China	KR	Republic of Korea	PT	Portugal		
CU	Cuba	KZ	Kazakstan	RO	Romania		
CZ	Czech Republic	LC	Saint Lucia	RU	Russian Federation		
DE	Germany	LI	Liechtenstein	SD	Sudan		
DK	Denmark	LK	Sri Lanka	SE	Sweden		
EE	Estonia	LR	Liberia	SG	Singapore		

A TOOL FOR SOFTWARE DIAGNOSIS DIVIDED ACROSS MULTIPLE PROCESSORS

BACKGROUND OF THE INVENTION

This application claims priority of U.S. Provisional Patent Application No. 60/005,110, filed September 1, 1995, entitled "A Tool for Software Diagnosis Divided Across Multiple Processors," with Attorney Docket No. 014274-000200.

This invention relates to software development and debugging environments. In particular, the invention relates to development and debugging environments wherein the software to be analyzed resides in a processor system distinct from but connected to another processor system running the debugging software's interface with the user.

In the development of software, the use of a debugging program (or "debugger") is not uncommon to analyze software to be developed or known to have a bug. This is the test software. One of the primary functions of such a debugger is to allow the user -- typically a development or diagnostics engineer -- to view the memory as modified by the execution of the test software.

As C and C++ are the programming languages currently enjoying the greatest acceptance among programmers in the art, this invention is described in C and C++. Of course, any programming language capable of the concepts discussed herein can realize this invention and its benefits.

The problems of debugging software are amplified in certain respects when the processor system on which the test software resides (the "target system" or "target") has insufficient resources to support a debugging environment. The usual critical resource is memory. In these situations, a substantial portion of the debugging environment can be displaced to a separate remote system, the "host system" or "host," and the target and host systems communicate by a communications link. The software on the target communicating over the link is the debugging agent, and that portion residing on the host is the debugging server.

The resolution of the limited memory problem by displacement results in other problems: communicating using the limited bandwidth of the communications link, for example. The bandwidth limitation is particularly acute when mass or high-speed data throughput is not the primary function of what becomes the communications link for development or debugging purposes or when complex and/or large data structures such as linked lists or tables must be retrieved by the debugging server from the debugging agent. In prior art systems, a significant portion of the bandwidth expended on retrieval of data from the target is simply the transmission of data requests from the server to the agent. For example, the retrieval of the linked list linked_data_list of the following data structure

```
structure datum
{
     int   * first_element;
     int     second_element;
     datum *datum_pointer;
} *linked_data_list;
```

requires first requesting and retrieving the address of the datum structure at the head of the list (*linked_data_list), then requesting and retrieving sizeof(structure datum) bytes from that address, then requesting and retrieving sizeof(structure datum) bytes from *(linked_data_list->datum_pointer), and so on. Any reduction in the amount of data transferred as data requests or the amount of data transferred in response to data requests would be a relief from the constraints of the narrow-channel communications link.

Another prior art method for retrieving linked_data_list would be to have a special service, GET_linked_data_list, built into the debugging agent. Thus, when the debugging server wants to look at linked_data_list, it simply invokes the GET_linked_data_list service. However,

BEST AVAILABLE COPY

given the variety of variable types typically found in an operating system and the additional number in the attendant application code, such a scheme does not bode well for minimizing the footprint of the debugging agent on the target memory. The alternative of restricting the freedom of the programmers in declaring variable types is not attractive either.

Even with displacement, the debugging agent can still consume a not insignificant amount of the memory of the target computer. An enduring design struggle is to provide sufficient services in the debugging agent to create a useful and even capability-laden agent while attempting to keep memory consumption to a minimum. A part of this struggle is whether to include services which are not frequently used but are extremely helpful if present when needed.

The uploading of data to the host computer can consume a major percentage of the available bandwidth on the communications link. Typically, the debugging server lacks the intelligence to ask whether the data being transferred is duplicative or otherwise unnecessary. Accordingly, there is a need for a debugging server which screens requests for data of the target and thereby reduces the amount of traffic on the communications link. There is also a need for a debugging server to screen data destined for the target and thereby reduce the amount of traffic on the communications link.

Many device-controlling systems provide only the communications interfaces necessary to control the device or devices of interest. One justification for so limiting the interfaces is cost. However, in a target/host development configuration as described above, at least one communications interface is given over to supporting the target/host configuration. Accordingly, the reallocation of communications interfaces disturbs the system under test, and the test scenario loses a certain amount of verisimilitude or verifiability. It would be preferable to connect all the controlled devices as well as the host system to the target machine -- even in configurations where each of N controlled devices requires a respective one of N total communications interfaces.

Some systems do not incorporate any means for communicating with the external world during normal operation. Nonetheless, a developer would like to debug in situ these and other similarly situated systems (e.g., systems where interrupting (any of) the fully allocated communications interface(s) is not preferable). A technique to enable the developer to debug these severely resource-limited systems is needed.

Prior art exists to establish a communcation path to a resource-constrained system as described. These devices include the NetRom product available from Applied Microsystems Corporation of Redmond, Washington. Typical of its class, the NetRom product connects into a ROM socket on the target side and provides a standard ethernet or serial interface to the host. However, to knowledge, such devices have been used primarily to replace programmed ROMs in unstable systems. Reprogramming the software available via a NetROM is easier than reprogramming an actual ROM.

Accordingly, an object of this invention is to create a debugging environment which has a smaller footprint in the memory of target systems.

Another object of the invention is to minimize the size of the debugging agent in the target memory while still offering all of the capabilities a software developer expects from a full-featured debugging environment.

An object of the invention is the addition of intelligence to the host-side software in order to reduce the consumption of available bandwidth by reducing both the number and size of target/host transactions.

Another goal of the invention is the addition of intelligence to the host-side software in order to obviate certain target-side intelligence and thus free the memory the target-side intelligence would consume.

Yet another goal of the invention is, in a target/host development environment, to connect all controlled devices as well as the host system to the target machine, particularly when each of N controlled devices requires a respective one of only N available communications interfaces.

Another object of the invention is, in a target/host development environment, to connect to the host a target machine with no conventional communications interfaces available.

Still other objects and goals of the present invention will be obvious to a person of ordinary skill in the art on reading the background above and the disclosure below.

SUMMARY OF THE INVENTION

The present invention is apparatus and methods coupling a first, target processor to a second, host processor. The target processor runs a debugging agent while the host processor runs a debugging server application which interacts with its counterpart, the debugging agent. The two processors are coupled by a communications link.

In one embodiment, to retrieve data from the memory of the target processor to the memory of the host, the host processor generates a program in an interpreted language to read the data from the memory of the target processor and communicates that program to the target processor. The target processor interprets the program, reading the data and communicating it over the communications link to the host processor. The program and/or the data communicated can be compressed for transmission and decompressed on reception. The advantages of the apparatus and method include reduced traffic on the limited bandwidth of the communications link and the avoidance of dedicated target-based services of limited use.

In another embodiment, the application to be developed and the debugging agent share libraries and provide debugging services from the shared libraries. Application and debugging code are partially or fully linked together. User demands are satisfied by executing code in the core common to the application and the debugging agent. In a preferred embodiment, a copy of the core is placed on the target machine while a copy of the symbol table is placed on the host. The debugging agent is then scalable beyond the services in the core by default. The amount of redundancy between the debugging agent and the application is at the discretion of the application developer. The developer controls the impact of the debugging agent on target memory,

In another embodiment, in order to debug applications, the program text of the debugging agent is supplemented to provide debugging services as the user demands them. Scalable debugging agents fundamentally extend the range of scalability beyond that of traditional target/monitor approaches.

In one embodiment, the debugging target and server cooperate to read memory of the target computer into the memory of the host computer coupled to a user input device. When a user (e.g., a user program displaying a thread of execution by continually displaying the relevant stack) issues first and second commands to read the memory of the target computer, the host computer determines the time differential between the sequential commands and reads the memory in response to the second command only if the time differential is greater than a predetermined value.

In another embodiment, the host computer is again coupled to a user input device. When a user directs a modification of the memory of the target computer, the host computer generates a script for moving data extant in the memory of the target computer and/or copying over data to the memory of the target computer so as to recreate the modification specified by the user. Where the size of the modification is small compared to the size of the total data, communicating the changes rather than the total modified data conserves the limited bandwidth of the communication means coupling the two computers together.

In yet another embodiment, the target processor is adapted for coupling to and controlling an external device. The target processor has a number of communications channels needed to operate in its final configuration, including one which controls the external device. The host processor uses this one communications channel for coupling to the target device and uses one of its communications channels to couple the target processor to the external device.

In a further embodiment, the invention couples the host computer and the target processor at the debugging target/server level by means of an in-circuit emulator. One such in-circuit emulator is a ROM emulator.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 illustrated a cross-platform environment according to a preferred embodiment;

Figure 2 illustrates a host/target system with a host-resident cache;

Figure 3 is an abstract diagram of a host/target system and their respective memories;

Figure 4 illustrates the levels of software supporting an embodiment of a virtual I/O channel;

Figure 5 illustrates a target and host connected by remote communications; and

Figure 6 illustrates the architecture of the target debugging server.

DESCRIPTION OF THE PREFERRED EMBODIMENT

TABLE OF CONTENTS

1. Overview 7
2. Gophers 9
3. Debugging Services on Demand 16
4. Shared Run-Time Debugging Services 20
5. Quantized Streaming Differential Cache 21
6. Host-Based Target Memory Management 24
7. Virtual I/O 25
8. In-Circuit Emulation 27
9. World-Wide Access 28
WHAT IS CLAIMED IS 30

1. Overview

The processor system most likely to benefit from the invention disclosed herein is the cross-platform for software development or software debugging. Figure 1 illustrates a 8

BEST AVAILABLE COPY

cross-platform environment 100 incorporating the invention. In such a cross-platform environment, the debugging software is divided in two. The first portion of the debugging software, the debugging agent 110, resides on a target computer 120 along with the software 130 undergoing development or debugging. The second portion of the debugging software, the debugging server 140, resides on a host computer 150 which provides an interface (here Tornado™ Launcher 160) to the developing or debugging user. The target and host computers are connected by means of a communications link 170 which can be serial, parallel, LAN, WAN, or any other form of I/O or hardware-assisted emulator or romulator.

The preferred operating system for the target is the VxWorks™ operating system 130b, available from the assignee of this patent application. Communications between the host and target are by means of the ONC RPC protocol, which can be implemented on virtually any transport layer. The host and target respectively support the following services (described from the host's vantage):

- ping a target
- connect to the debugging agent
- disconnect from the debugging agent
- set the debugging agent mode
- read a target memory region
- write a memory block to the target
- fill a target memory block with a pattern
- move a target memory block
- compute the checksum value of a target memory region
- turn protection on or off for a target memory region
- update the target instruction cache
- search for a pattern in the target memory
- create a context on the target
- destroy a context on the target
- suspend a context on the target
- resume a context suspended on the target

- get values from target registers
- set values of target registers
- write data to a target virtual I/O channel
- add an eventpoint on the target
- delete an eventpoint on the target
- get an event from the target
- continue a context suspended on the target
- single step a context on the target
- call a function on the target
- call a function in the agent
- evaluate a gopher string on the target
- check if an event is pending on the target
- clear an event pending on the target
- call a service on the target
- free the memory used by the returned structure

Each of these services is described in the Tornado™ API Guide, available from the assignee of the instant patent application.

2. Gophers

In order to reduce traffic on the communications link between the target and host, a system uses a first embodiment of the invention: Gophers. Gophers are interpreted programs for efficiently retrieving data from the memory of the target to the memory of the host. Gophers are particularly useful when the data to be retrieved has a complex structure.

In a preferred embodiment, the Gopher language has only one state indicator, a programmable variable referred to as the "pointer." Also, the Gopher language has only one I/O device, an abstract "tape." All Gopher actions are interpreted relative to the value of the pointer, and all output is via the tape.

The Gopher language also includes the following syntactic elements:

Table I:

Gopher Language Syntactic Elements

Directive	Action
n	Set the pointer to the integer value *n*
[+-]*n*	Increment or decrement the pointer by the integer value *n*
!	Write the pointer to the tape
*	Replace the pointer with the value pointed to by the pointer
@[bwl]	Write the value pointed to by the pointer to the tape, and advance the pointer. "B," "W," and "L" indicate that the size of the value is 8, 16 or 32 bits, respectively. The pointer is advanced by the size of the value written. ("L" is the default.)
$	Write the (zero-terminated) string to which the pointer points to the tape.
{ . . . }	Create and use a local copy of the pointer to repeatedly evaluate the Gopher string between the curly braces until that pointer is zero. Upon completion, the local copy of the pointer is destroyed and the original is unchanged.
< . . . >	Create and use a local copy of the pointer to evaluate the Gopher string between the angle braces once. Upon completion, the local copy of the pointer is destroyed and the original is unchanged.
(*n* . . .)	Create and use a local copy of the pointer and evaluate the Gopher string inside the parentheses *n* times.

A Gopher can be used to retrieve, for example, the linked list ifnet containing most of the information for network interfaces in the preferred target operating system. The structure ifnet is declared as follows:

```
struct ifnet {
  char*if_name;       /* name, e.g. ''en'' or ''lo'' */
  shortif_unit;       /* sub-unit for lower level driver */
  shortif_mtu;        /* maximum transmission unit */
  shortif_flags;      /* up/down, broadcast, etc. */
  shortif_timer;      /* time 'til if_watchdog called */
  int if_metric;      /* routing metric (external only) */
  structifadder *if_addrlist; /* linked list of addresses per
                                 if */
/* Some members of this structure have been omitted here for
simplicity. */

     ...
/* generic interface statistics */
  int if_ipackets;          /* packets received on interface */
  int if_errors;            /* input errors on interface */
  int if_opackets;          /* packets sent on interface */
  int if_oerrors;           /* output errors on interface */
  int if_collisions;        /* collisions on cama interfaces */
/* end statistics */
  structifnet *if_next;
  *ifnet_ptr;
}
```

An algorithm to traverse the ifnet list might appear as follows:

```
p = ifnet;
while (p != NULL)
{
     print p;
     p = p->if_next;
}
```

and could be implemented at the host-target interface as a target-implemented service or by constructing a command file or application which repeatedly uses the target memory read service. The equivalent Gopher, however, for the algorithm is:

```
ifnet { ! + 84 * }
```

This Gopher program is deconstructed as follows: *ifnet* is the integer address of the ifnet pointer, which can be obtained from the symbol table by means well-known in the art or by querying the debugging server 140 regarding symbol table 180. <ifnet> sets the value of the Gopher pointer. The left brace creates a local copy of the pointer and starts a Gopher loop. The Gopher interpreter will execute the subprogram "! + 84 *" until the local copy of the pointer is not NULL (zero) when the right curly brace is reached. The syntactic element ! writes the value of the (local) pointer to the output tape, and "+84" then increments the (local) pointer. The "*" expression causes the (local) pointer to be replaced by the quantity to which the (local) pointer currently points. A link is thus followed. The "}" delimits the end of the loop, which is terminated when the address to which the (local) pointer currently points is NULL.

In all Gopher expressions using matching delimiters (i.e., "{ . . . }," "< . . . >" and "(. . .)") the Gopher evaluator saves unchanged the pointer in effect outside the delimiters and creates and uses another pointer for use during the execution of the delimited expression. The evaluator restores the outside pointer when the closing delimiter is executed. While no code follows the closing delimiter in this example, there may, of course, be occasions when such code will exist.

Also, all writes are terminated by white space. This allows the contents of the tape to be processed by a subsequent filter having no additional knowledge of the internal characteristics of the data on the tape.

Note that Gophers have no symbolic capacity and do not understand the layout of structures. The "+84" expression was determined from inspecting the size of the elements constituting the ifnet data structure. This is a cost of the language independence of Gophers. Advantages include minimal syntactic elements and compactness.

The above example returns a list of network interface descriptors. A more sophisticated Gopher can extract desired fields from each of those data structures.

The fields of interest in the ifnet data structure are deemed to be the name of the interface, the sub-unit for the lower-level driver, the maximum transmission unit, the flags, the routing metric and the five packet-statistics, respectively, the if_name, if_unit, if_mtu, if_flags, if_metric and if_ipackets through if_collisions members.

With the Gopher's pointer pointing to the first byte of an ifnet data structure, it also points to the address of the string naming the interface. A Gopher program string to dereference the pointer and write the string addressed follows:

```
<*$>
```

The left angle bracket introduces a subcontext which saves the value of the outer pointer. After execution of the ">," the pointer still points to the top of the data structure, despite the dereferencing and advancement of the local pointer caused by the pointer-replacement and string-write actions.

To fetch the next four fields of interest, with the pointer still pointing to the if_name field, the following Gopher program string advances the pointer over the four bytes of the if_name pointer ("+4") and then writes the value pointed to by the now-advanced pointer ("@"):

```
+4 @w @w @w +2 @
```

Because if_unit, if_mtu and if_flags are shorts, the value write actions are modified with "w." The Gopher program string "+2 @" advances the pointer over the watchdog timer element and writes the external metric, a four-byte quantity.

The packet statistics are consecutive within the data structure and the following program string suffices to retrieve them:

```
+48 @@@@@
```

Thus, the entire Gopher program to retrieve the ten fields of interest is:

```
<*$> +4 @w @w @w +2 @ +48 @@@@@
```

Assuming one byte per character in the Gopher language, the above program consumes only 32 bytes -- 24 bytes if the unnecessary white spaces inserted for human form factors are removed. Only one remote procedure call to execute this 24-byte Gopher program is necessary to retrieve the desired data, and only the 30+ bytes actually desired are transferred.

By comparison, in order to limit the number of RPCs, the prior art can read large blocks of data of, say, sizeof(struct ifnet) bytes or more. Eighty-four bytes are transferred when only 30 bytes are desired, and some extraneous number of bytes from the large blocks will be transferred as well. Also, the number of RPCs increases as large blocks are read as necessary to retrieve the NULL-terminated name of the interface, and the number of extraneous bytes transferred also increases.

On the other hand, in order to limit the number of bytes of data transferred in response to requests, the prior art can execute a series of RPCs, one for each non-contiguous block of ifnet data structure members desired.

The superiority of Gophers is highlighted when the retrieval of the above 30+ bytes from each of a number of linked ifnet structures is considered. For N such structures, the prior art RPCs increase by a factor of N, and the extraneous data transferred increases by the same factor. In starkest contrast, the Gopher program increases by a mere <u>one byte</u> from

```
        ifnet < <*$> +4 @w @w @w +2 @ +48 @@@@@ >
```

to

```
        ifnet { <*$> +4 @w @w @w +2 @ +48 @@@@@ * }
```

Further, with Gophers, the number of RPCs remains at the lowest number possible: one. The significance of this cannot be overstated. As a person of ordinary skill in the art will appreciate, different levels of the communications standard supporting the remote procedure calls (e.g., the International Standards Organization ("ISO") Open System Interconnection Reference Model ("OSI")) impose overhead, and the bandwidth which the cumulative overhead consumes can be truly large. An ability to keep the number of RPCs to the absolute minimum is necessary to minimize bandwidth consumption.

The prior art can minimize the number of RPCs (and thus the bandwidth consumed in sending data requests), or it can minimize the amount of data requested (and thus the bandwidth consumed in sending data back), but the prior art does not minimize both the number of RPCs and the amount of data transferred in response to those RPCs. With its' ability to not merely reduce but truly minimize the consumption of bandwidth, Gophers represent a clear and major improvement over the known art.

The debugging agent includes an interpreter for the Gopher language. Execution time of the interpreted Gopher is greater than that of a compiled and linked Gopher would be but still represents only a very small fraction of the overall transfer time. The Gopher interpreter itself is very small. In fact, the Gopher interpreter available from the assignee of this patent application is smaller than the amount of the memory that would have been required by target routines to gather the data of only the basic system objects.

A compression-decompression scheme can reduce still further the bandwidth which the transfer of a Gopher program or of the data in response to the Gopher program consumes. If, for example, the linked list in interest was a list of buffers known to be mostly zeroes, a compression-decompression scheme initiated either by the server or the agent and communicated to the other would still further reduce the load on the communications link.

The Gopher language is compact to reduce communication overhead and is accessible at higher level interfaces within the development environment.

In one embodiment, some or all Gophers are pre-generated based on the information available from the symbol table regarding the data items of interest. Thus, when a user directs the retrieval of a particular data structure, the server need only recall and transmit the pre-generated Gopher produced in anticipation of that direction.

3. Debugging Services on Demand

With the debugging environment divided between the agent on the target and the server on the host computer, still further reductions in the size of the debug environment in the target memory are nonetheless desirable. Accordingly, this invention provides a scalable debugging agent.

The debugging server 140 has the capability of incrementally loading object modules into the target system 120. To achieve this end, the debugging server 140 uses the symbol tables contained in every object module to build a system-wide symbol table of functions and variable names loaded in the target system 120. The debugging server 140 adds names from system and application modules 130 alike to the system symbol table. Indeed, the debugging server adds names from the debugging agent 110 itself to the system symbol table.

Fig. 6 illustrates the architecture of the target debugging server 140. As shown in Fig. 6, the debugging server 140 maintains a copy 610 of the debugging agent 110 on the host (or, at least, a copy of the symbol table of the agent). This copy 610 allows the debugging server 140 to read and analyze the agent 110's core file to determine what symbols are currently available (resolved) in the agent core. Without a single upload, the debugging server 140 is "bootstrapped" with the core information. The debugging server 140 can immediately execute user commands entered through a shell 160 and can perform other functions -- constrained only by what services have been linked into the core.

Correspondingly, the host software 160 will typically include a shell 160d or 160e (with underlying hardware, not shown) or other mechanism for communicating with the user. The host accepts commands on this user input device from the user. (A device for user input is most typically a keyboard or mouse. However, a disk, tape or other medium file containing commands previously stored by the user can also be a device for user input.)

Some of these commands the software 160 can satisfy directly; others may require supplemental information or service from the debugging server 140. Still other commands may require information or services which only the agent 110 can provide. For these commands requiring information or services from the agent 110, the debugging server 140 breaks the user command (as relayed by the software 160) into the necessary transactions with the debugging agent 110.

Before the debugging server 140 requests a service of the debugging agent 110, the debugging server 140 checks the host-resident copy 610 of the debugging agent core to determine whether the symbol(s) necessary to provide the service are resolved in the debugging agent 110.

If the symbols which provide the desired functionality are resolved, then the debugging server 140 can simply request (e.g., via RPC) that the debugging agent 110 provide the service. Where the symbols providing the requested service are not resolved in the agent 110, the debugging server 140 reads a memory device (not shown) to retrieve the object module containing the text, data, bss or literal sections (depending on the unresolved symbols present and the format of the object module) to which the unresolved symbols refer.

In one embodiment, the host 140 then relinks the debugging agent image on the host 150 to include the new sections, resolving the symbols of interest, and produces a list of differences between the old agent image and the new agent image. Using the differences list, the debugging server 140 then directs the debugging agent 110 through a series of target memory modifications (memory moves, writes, etc.) to reproduce in the target memory the re-linked debugging agent core on the host.

In an alternative embodiment, after reading the memory device to retrieve the sections to which the unresolved symbols refer, the debugging server 140 directs the debugging agent 110 to accept the sections and relink itself. This second embodiment is less preferable in that it requires a larger debugging agent 110 with sufficient intelligence to relink itself.

The result of both of the foregoing embodiments is a debugging agent 110 with the symbols supporting the requested service resolved. The debugging server 140 then directs the debugging agent 110 to perform the desired service, thereby providing the information or service which the host software 160 requires and performing, directly or indirectly, the user's command.

In a preferred embodiment, the debugging server can direct the agent to discard the newly loaded code after use.

A debugging agent that is scalable allows for an agent equipped to handle one environment, say, a production environment, to be extended when necessary to support another environment, say, a development environment or an in-the-field debugging environment. A scalable debugging agent is equipped initially with an optimal subset of the set of all services which it could offer on a target of substantially unlimited resources, and the subset is extendable on demand to offer any of the full set of services. In a preferred embodiment, scaling of the agent is automatic, occurring in a manner transparent to the user.

The primary services which a scalable debugging agent 110 offers are target memory write and target service calls. These primary initial services are sufficient to support the corresponding server 140 on the host.

As mentioned above, a program 160 with a user interface for debugging the target software resides on the host. In the preferred embodiment, this program 160 is the remote source-level CrossWind™ debugger 160d available from the assignee of this patent application. The CrossWind™ debugger 160d extends the GNU Source-Level Debugger (GDB) with a graphical user interface and with a comprehensive Tcl scripting interface which allows developers to create sophisticated debugger macros and CrossWind™ extensions. The CrossWind™ debugger 160d permits a user to spawn and debug tasks on the target. The user can also debug already-running

BEST AVAILABLE COPY

tasks spawned with a runtime shell. Debugging includes setting system and task-level breakpoints.

The memory device to which the host debugger 160d refers for the original debugging agent core file or for the additional sections can be a conventional host-based RAM, ROM, disk, user input device or other host-based memory device. Alternatively, the host debugger program 160d can query the agent 110 itself for this information. (The agent 110 is typically loaded from ROM.) Accordingly, the agent 110 contains the origins of itself and its scaling object modules. Querying the agent 110 allows use of agents 110 other than those provided by the assignee of the patent application because no assumptions are made about the construction of an agent 110. The host explicitly asks the agent 110 for information as necessary.

As a precautionary measure, the host may request the agent 110 to provide a checksum of its text region. The host then compares this checksum against a checksum of the text region of the core file specified by the agent 110. One reason that the two checksums might differ is that the core file on the host may have been updated independently of and after the writing of the agent to the target ROM.

The types of services which can be thus incrementally loaded are limited only by the services which must initially reside in the target in order to enable the loader and user interface in the server. The following agent services are good candidates for dynamically scaling into the agent 110: Serial line testing, breakpoint support, context (task) control, event support, host exception notification, external context function calls, gophers, memory services, (other than write and move) register access and virtual I/O support.

Of course, if dynamic scaling as described above is applied to even the most mundane agent services, the size of the debugging agent 110 in the target memory is guaranteed to always be at a minimum. On the user's first invocation of a particular command, the debugging server 140 can download the supporting code -- transparently to the user -- to the agent

110. No (or minimal) services will exist in the agent 110 which the user did not actively need and thus no (or minimal) excess code will exist in the target memory. The debugging agent 110 is essentially customized for the particular debugging session.

In order to avoid the target 110's reserving the maximum amount of memory a scalable agent may ever need just in case the agent grows to that size, the invention includes a method for sharing memory between the debugging agent and the operating system software or other application of the target. In a less complicated form, the debugging agent is allowed to grow, consuming more and more memory. In a more sophisticated form, the debugging agent both grows and shrinks, consuming and releasing memory in the process.

4. Shared Run-Time Debugging Services

The invention includes another method for reducing the size of the debugging agent 110 in the memory of the target computer 120. Typically, application code 130 relies upon a library of functions provided with the application development environment. Among other things, these standard libraries obviate the need for the program developer to program and re-program commonly used routines, and they can make the interfaces to operating system services more programmer-friendly.

The conventional wisdom is that a debugging agent 110 (for that matter, debugging tools in general) and the program 130 to be debugged should not share libraries. There may be an interaction between the agent 110 and the program 130 which creates or, more subtly, hides a problem. However, these standard libraries can be very large and duplicating them for both the agent 110 and the application 130 makes the amount of target memory occupied even greater. Accordingly, sharing standard libraries between agent and application can offer a material benefit in conserving target memory.

In one embodiment, this invention provides a simple mechanism to determine if code is to be shared or not. When incrementally adding new application code to the target system

120, the host-based loader 620 merely references the most recently defined symbol for the purpose of symbol resolution. Duplicate entries in the system symbol table are not rejected. In this way, at his discretion, a user may download redundant copies of services provided by the agent in the core image or may link against the resident services.

Such an approach need not be limited to the entirety of a library. Even the sharing of a single routine within a resident library can be avoided. Consider the situation where both the agent and an application need functions f() and g() which exist in a library which could be shared. The application can be linked such that function f() comes from the library shared with the agent. Function g() can be supplied in the application's source code, and incrementally downloaded to the system before the remaining body of application code is incrementally downloaded. The preferential reference of the loader to the more recent definition of the function g() will resolve the reference to the unshared definition of g(). In prior art, such control over the degree of shared runtime code has not been possible.

When the scalability of the debugging agent 110 is combined with shared libraries, the advantages are geometric.

5. Quantized Streaming Differential Cache

In responding to user requests, the debugging server classically turns to the debugging agent, asking for data to be uploaded. The debugging server then uses the uploaded data to produce a response to the user. If the user repeatedly requests the same data, a conventional server will repeatedly request the data of the agent and accordingly prepare the responses to the user.

According to the present invention, the copies of portions of the target memory which have been uploaded and now reside in the memory of the host can be likened to a data cache, with the original data residing on the target computer.

Further, as with all data caches, the question arises: When are reads satisfied from the cache and when must the read be satisfied from the original data? Conventional debugging servers ignore the caching potentials and always satisfy data read requests from the original data on the target. The proffered justification is the need to have the latest information from the target. However, most of the reads performed to satisfy these requests are not performed in a locking mode wherein the accuracy of the data being read is assured by locking out any code which could modify the data during the reading. Even with reads performed with locking, there is no guarantee that the modifying code will not change the original of the read data after completion of the locked read but before its presentation to the user. There is no guarantee that what the user sees will be the actual state of the target system. In short, conventional debugging servers or agents cannot insure that data presented to the user is accurate and sometimes cannot insure that data presented is internally consistent.

It is preferable to use Gophers performing locking reads to upload data structures associated with system objects. This method insures at least the internal consistency of important objects.

Recognizing that data reads to satisfy user requests may not produce consistent or accurate data, a system incorporating an embodiment of the invention maintains a variable memory register 230 holding a threshold age. Under certain conditions described below, the system avoids consuming bandwidth with data reads.

Figure 2 illustrates a host/target system 220 with a host-resident cache 210. The host software 140 satisfies data requests from the cache 210 if the data in the cache 210 is younger than the threshold age. If the threshold age is some predetermined value, such as zero, the server 140 always satisfies data requests by reference to the agent 110. Preferably, the user chooses the threshold age for the particular situation. Experiential data suggests that a threshold age of one second suffices for day-to-day situations.

In another embodiment, the server maintains individual age thresholds for respective blocks of data in the host cache. Accordingly, data known or deemed to be volatile can be refreshed at a greater rate than data known or deemed to be more static. In both of these embodiments, the flexibility is in the hands of the user.

Another issue associated with caching is writing back: modifying the original when the copy has been modified. The debugging server cache 210 disclosed herein is "write-through," the data on the target 120 is updated essentially immediately after the copy on the host 150 is modified. The justification here is that the user always intends to modify the data actually on the target. Indeed, the user may not even be aware that the host software is performing any caching.

In certain circumstances, the required writing back can be significantly streamlined. When the original data in the target memory is unchanging (and, therefore, consistent with the host copy), only the writing back of the changed portions of the data is necessary, rather than writing back the entirety of the cached data. Thus, where cache blocks are, say, a kilobyte in size and the change affects only tens or small hundreds of sequential bytes, an advantage lies in instructing the debugging agent 110 to modify the affected tens or hundreds of bytes rather than instructing the agent 110 to re-write the entire kilobyte.

Another example is the reloading of a software application or module. Usually, when an application is changed to remedy some bug, only a small percentage of the application text changes (to add or delete a few lines of code) when re-compiled. Yet in the prior art, when this application is loaded onto the target, the loading consumes bandwidth as necessary to transfer the entire file. However, the invention views the application as a stream of data and the typical recompilation as the insertion and/or deletion of a small portion of the stream. When downloading a new stream of memory to the target 120, the stream is compared with the copy of the original target contents cached on the host 150, and the target 120 is instructed to adjust its memory to slip contents forward or backward to agree with the new module being constructed. The host 150 then transfers new data, if any, to the target 120 to fill the newly vacant or other memory spaces.

Relocation fields within the stream of data complicate the algorithm because they will almost always differ. However, the list of locations of the fields for both the original and the new stream of data are known beforehand. If the module is re-linked after transfer, these differences are inconsequential.

6. <u>Host-Based Target Memory Management</u>

In some cases, the development tools may require resources from the target in addition to those required by the application software on the target. These additional resources can be, for instance, sophisticated services which require exclusive access to an allocated portion of target memory. Among the most sophisticated of these memory-related services is dynamic memory management.

Figure 3 is an abstract diagram of a host/target system 300 and the respective memories 310 and 320. Systems embodying the invention take into account the fact that the memory 310 of the host system 150 is usually much larger than the memory 320 of the target system 120 and, therefore, the host system 150 can provide storage for many house-keeping data structures used in the management of the target-resident memory 320. With the intelligence (and code space) in the host 310 managing free lists of memory and other cumbersome data structures, the debugging agent 110 need only provide services such as move, copy and fill.

The two data structures typically supporting a memory manager are a free memory list 330 and the allocated list (not shown). (The latter is sometimes excluded for efficiency.) In one embodiment, the debugging server 140 will allocate host-based memory for these data structures and thus realize zero-overhead performance when allocating target memory. The free list 330 is a linked list of free blocks with each link in the list containing the address of the corresponding contiguous free block of memory on the target and the size of this block. The list 330 is not ordered, and allocation is achieved with a traditional "first-fit" search of the list. The third data structure according to this embodiment is an AVL-tree or binary tree with nodes containing target memory block information sorted within the tree from lower target address to higher target address. This data structure is traversed whenever target memory is deallocated, in order to find the host-resident node that corresponds to the target block to be deallocated. The use of a b-tree limits the worst-case search to log N, where N is the number of allocated blocks in the tree.

In prior art, the target debugging agent manages such data structures. By virtue of this invention, target systems incur less overhead in target memory allocation, thus saving resources for application use. Another virtue of this invention lies in that host-to-target protocol requests related to memory allocation are greatly reduced limiting the intrusion of host tool memory allocation on the real-time execution of a target system application.

7. Virtual I/O

Consider the scenario wherein the target 110 is a device-controlling processor. It has exactly two communications interfaces, one to receive input from a sensor and the second to control a device based on that input. The operations of the device affect the characteristic read by the sensor. The memory of the target is limited so that a target/host development environment is preferable or required. The question becomes how to connect the host when both communications ports are necessary for operation of the target.

A system incorporating the invention maintains virtual I/O (VIO) channels. A VIO channel can be implemented as a pseudo-tty, a stream as in UNIX®, an underlying device, etc. The host 150 redirects communications intended for the controlled device to a communications channel of the host machine, to which communications channel the controlled device is connected. The target machine is thus in communication with the sensor, the controlled device and the host computer even though the target machine has only two communications interfaces.

On the target 110, the operating system 130b supports a virtual I/O device driver 190 to provide target applications 130a with I/O system access to the virtual I/O facility according to the standard system call protocols of the target operating system 130b. In a UNIX® - or Posix-like operating system, a first open of a VIO channel is as follows:

```
/* initialize virtual I/O driver */
/* this routine should only be called once */
.
.
.
if (VIO_DRIVER_STATE != OK)
    return(ERROR);
/* create virtual I/O channel 0 named "/vio" */
/* with a 512-byte read buffer */
/* and a 512-byte write buffer */
if (VIO_DEVICE_CREATE("/vio", 0, 512, 512) != OK)
    return(ERROR);

/* open a file on VIO channel 4 */
if ((fd = open("/vio/4", O_RDWR, 0)) == ERROR)
    return(ERROR);
```

The host 150 associates a VIO channel with any one of a wide variety of devices, including displays, files, raw devices, streams, etc. A host application reserves a virtual I/O channel using a system call. As a VIO channel number is used by both the server and agent to determine the source or destination of particular data, the host and target applications must agree as to the channel number. The host application typically then redirects the reserved VIO channel to an actual device on the host, using a system call to control the VIO channel. Following is a code sample redirecting VIO channel 4 to a file:

```
/* open host file */
if ((fd_host = open ("/tmp/testfile", O_RDWR, 0)
    == ERROR)
    return(ERROR)
```

```
/* send VIO redirection command of channel 4 to
/tmp/testfile */

if (wtxVioCtl (hWtx, 4, VIO_REDIRECT, fdHost != WTX_OK)
    return (ERROR);
```

Any data which the debugging agent 110 reads from or writes to "/vio/4," i.e., VIO channel 4, the target server 140 will redirect from or to "/tmp/testfile."

On the host side, the target server 140 implements the mechanics of VIO channels. The debugging server 140 maintains the VIO channel pool, allocating free VIO channels and freeing VIO channels on request. As illustrated above, the target server 140 also redirects data as requested.

FIG. 4 illustrates some of the levels of software supporting an embodiment of a virtual I/O channel (from the target to the host). The host 400 includes a virtual console 410 in communication with the target server software 460. The target server 460 is in communication with the corresponding debugging agent software 420 residing on the target 440. Also on the target 440 is the virtual I/O device driver 430, which is preferably a part of the operating system residing on the target 440. Application software 450 makes the standard operating system calls (e.g., write() on a UNIX®-like operating system) to the virtual I/O device driver. In FIG. 4, the virtual I/O device driver 430 instructs the debugging agent to transfer a buffer containing "Hello world\n" to the debugging server, which then directs the display of the string in the Xterm window 450 on the host 400.

8. In-Circuit Emulation

Consider targets with severely limited hardware resources. Providing a meaningful low cost development environment for this class of targets has historically proven difficult to impossible. For example, consider a target without the critical resource of a communications channel to the host, typically an ethernet or serial communications device. With no means of communication between the target and the host, interacting with the target during development of an embedded application is apparently impossible.

The invention exploits the power of ROM emulators and in-circuit emulators in general in a development environment, connecting a host-based development environment to a target-resident debugging agent in a resource-constrained environment. The advantages of such an approach in so connecting a debugging agent to the host server are two-fold. First, the ROM emulation provides a full-featured debugging environment to those targets in which no such environment could have existed in prior art.

Second, the intermediated communications device now provides the communications and off-loads from the target environment a significant portion of the debugging agent dedicated to communications. As mentioned above, the RAM of the target is often a critical resource. Any technique that reduces the footprint (i.e., storage requirements) on the target of the debugging environment is highly desirable, leaving more target RAM available for application needs. Where a NetROM provides the communications, it reduces the role of the debugging agent and in turn effects considerable savings in target RAM. The local agent program code to manage communications with the host would otherwise occupy the now conserved RAM.

9. <u>World-Wide Access</u>

An added benefit of the target/host configuration upon which the prior art has not capitalized is the realization that the host need not be in physical proximity to the target. Figure 5 illustrates a target 520 and a host 530 connected by means of remote communications technology 510. With the use of a WAN, e.g., the global telephone system with modem interfaces or a satellite communications system, the host 530 and/or target 520 can be anywhere on earth or near space where a port 540 onto the WAN (e.g., a telephone or satellite link) is available. With other known remote communications technology 510, the host 530 and/or target 520 can be anywhere that technology reaches. The communications technology 510 can be a switching network, allowing any host

530 with the address (e.g., telephone number) of the target modem to connect with the target 520.

When viewed as a representative of the target on the host, a host-based target server can broker requests for virtually any site. The Internet, with its suite of applications tracking who's who and where they are, is an excellent example of the remote communications technology 510. Its suite of remote services makes possible bringing up a shell, for example, on any target 520 with a known address. Further, a user can register a target server 530 globally, and an authenticated user can bring up any tool on the associated target 520 for the purpose of remote development, demonstration, technical support, etc. This paradigm of distributed World-Wide access to real-time execution platforms shifts the standard usage patterns of cross-development environments from one of single-user work environments to one of distributed development.

In one embodiment, the invention provides on-line access to all registered target systems 520 via their target servers 530 by means of a centralized control panel 160a (Fig. 1) summarizing all available targets and all available tools.

CONCLUSION

In using the above-disclosed methods and apparatus, the Assignee of this invention has succeeded in reducing the size of its debugging agents by an order of magnitude over the prior art. Typical agents according to the invention herein are approximately 20 kilobytes (Kb). By comparison, prior art agents are around 330 to 340 Kb. Consequently, the debugging agents disclosed herein are capable of running in a much broader range of target computers, particularly in those with memory less than 300 Kb.

Of course, where memory is not a constraint, the instant invention may be practiced with a single computer system running both the server and the agent.

WHAT IS CLAIMED IS:

1. A method for retrieving data from a first memory of a first processor, said method comprising:

coupling said first processor to a second processor by a communications link, said second processor having a second memory;

generating on said second processor a program in an interpreted language, said program for reading data from memory and for communicating said read data over said communications link;

communicating said program from said second processor to said first processor;

receiving said program on said first processor;

interpreting said program on said first processor, thereby reading data from said first memory and communicating said read data from said first processor to said second processor over said communications link; and

receiving said read data on said second processor and storing said read data in said second memory.

2. The method of claim 1 wherein said step of communicating comprises

compressing said program and

said step of receiving said program comprises

decompressing said program.

3. The method of claim 1 wherein said step of interpreting comprises

compressing said read data and

said step of receiving said read data comprises

decompressing said read data.

4. An apparatus for debugging an application program, said apparatus comprising:

a first processor having a first memory, said memory containing said application program and an interpreter of a program language;

a user input device through which a user can communicate a demand to examine a portion of said first memory;

a communications link; and

a second processor, coupled to said first processor by means of said communications link and coupled to said user input device, said second processor having a second memory containing a program adapted to, in response to said demand, generate a program in said interpreted program language, said generated program for reading said portion of said first memory and for communicating said read portion over said communications link to said second processor.

5. A method for debugging an application program, said method comprising:

coupling a first processor having a first memory to a second processor having a second memory by means of a communications link;

coupling said second processor to a user input device on which a user can communicate a demand for a debugging service;

running a debugging program and said application program in said first memory;

modifying, in response to said demand, the program text of said debugging program, enabling said debugging program to support said debugging service.

6. A system for debugging an application program, said system comprising:

a first processor having a first memory, said memory containing said application program and a debugging program adapted to perform debugging services;

a user input device on which a user communicates a demand for a debugging service; and

a second processor, coupled to said first processor and coupled to said user input device, said second processor having a second memory, said second memory containing a program adapted to interpret said demand and to modify the program text of said debugging program to enable said debugging program to support said debugging service.

7. The system of claim 6 wherein said program adapted to interpret is further adapted to communicate program code for performing said debugging service to said debugging program and said debugging program is adapted to bind said program code to itself and thereby become capable of performing said debugging service.

8. The system of claim 6 wherein said debugging program is further adapted to unbind said program code from itself when said debugging service has not been accessed within a predetermined amount of time.

9. A system for debugging an application program, said system comprising:

a first processor having a first memory, said first memory containing a symbol table for a core, said core being the linkage of application code for said application program and debugger code for a program for debugging said application program;

a second processor, coupled to said first processor, said second processor having a second memory containing a copy of said core uploaded from said first memory; and

a user input device, coupled to said first processor, on which a user communicates a demand for a debugging service, said debugging service performed by executing said debugging code in said copy of said core.

10. A method for debugging an application program, said method comprising:

coupling a first computer having first memory to a second computer having second memory;

linking together application code for said application program with debugger code for debugging said

BEST AVAILABLE COPY

application program, forming a core, said core having a symbol table;

placing a copy of said core in said first memory and a copy of said symbol table in said second memory;

interpreting a demand for a debugging service input on a user input device coupled to said second processor; and

performing said debugging service on said first processor by executing said debugger code.

11. The method of claim 10 wherein before said step of performing is performed, the following steps are performed:

checking said symbol table to determine whether code to perform said debugging service is already resident in said first memory;

uploading code to perform said debugging service when said checking determines that code to perform said debugging service is not already resident in said first memory and modifying said symbol table to reflect the fact that code to perform said debugging service is resident in said first memory; and

refraining from uploading code to execute said demand when said checking determines that code to perform said debugging service is already resident in said first memory.

12. A method for reading memory, said method comprising:

coupling a first computer having first memory to a second computer having second memory by means of a communications link;

running a program in said first memory, thereby modifying said first memory;

receiving a first command at a first time on said second computer, the performance of said first command requiring the reading of a portion of said first memory;

reading said portion of said first memory into said second memory;

receiving a second command at a second time on said second computer, the performance of said second command requiring the reading of said portion of said first memory;

reading said portion of said first memory into said second memory when the difference between said second and first times is greater than a predetermined value, and refraining from reading said portion of said first memory into said second memory when said difference is not greater than said predetermined value.

13. The method of claim 12 wherein said second step of reading further comprises

in any event, reading said portion of said first memory into said second memory when said predetermined value equals a second predetermined value.

14. The method of claim 12 wherein said second step of reading further comprises

in any event, reading said portion of said first memory into said second memory when said portion of memory is associated with a system object.

15. The method of claim 12 wherein before said second step of reading is performed, the following step is performed:

setting said predetermined value in response to a command on a user input device.

16. A system for inspecting memory, said system comprising:

an inter-processor communications link;

a first processor having first memory, a program running in said first memory and modifying said first memory;

an input device for receiving commands from a user; and a second processor, coupled to said first processor by means of said inter-processor communications link and coupled to said input device, said second processor having

second memory;

a register holding a time quantum;

an arithmetic logic unit; and

a CPU under program control, said program adapted to receive a first command at a first time by means of said input device, the performance of said first command requiring the reading of a portion of said first memory; to read said portion of said first memory into said second memory; to receive a second command at a second time by means of said input device, the performance of said second command requiring the reading of said portion of said first memory; to read said portion of said first memory into said second memory when the difference between said second and first times is greater than a predetermined value; and to refrain from reading said portion of said first memory into said second memory when said difference is not greater than said predetermined value.

17. A method for modifying memory, said method comprising:

coupling a first processor having first memory to a second processor having second memory by means of a communications link;

downloading a copy of first data from said first memory to said second memory;

generating second data in said second memory;

generating a script for modifying said first data to produce said second data;

transmitting said script to said first processor;

executing said script, modifying said first data to produce a copy of said second data in said first memory.

18. The method of claim 17 wherein said step of executing comprises

moving a portion of said first data from a first area in said first memory;

copying a portion of said second data to said first area.

19. The method of claim 17 wherein said step of executing comprises

overwriting a first portion of said first data with a second portion of said first data;

copying a portion of said second data to said first memory.

20. The method of claim 17, further comprising the step of

re-linking said copy of said second data in said first memory.

21. A system for managing the memory of a processor, said system comprising:

said processor having said memory;

a first application in said memory;

a second application in said memory, said second application adapted to develop said first application, said second application adapted to reserve free space from said memory and requiring memory management services for said free space;

a communications link;

a second processor, coupled to said processor by means of said communications link, said second processor having second memory; and

a third application in said second memory, said third application adapted to provide said memory management services for said free space over said communications link.

22. A method for managing the memory of a processor, said method comprising:

coupling said processor to a second processor by means of a communications link, said second processor having second memory;

running a first application in said memory;

running a second applications in said memory which reserves free memory, said second application for developing said first application, said second application requiring memory management services for said free memory;

running a third application in said second memory, said third application adapted to provide memory management services over said communications link;

requesting memory management services of said second processor, thereby generating instructions for said first processor;

managing the free memory of said first processor in response to said instructions.

23. An apparatus for analyzing a processor-controlled system, said apparatus comprising:

an external device adapted to be controlled by a processor;

a first processor having memory, said first processor adapted for coupling to said external device, said first processor having exactly the number of communications channels necessary for said first processor to operate in final configuration including running an application in said memory, thereby controlling said external device; and

a second processor having second memory, said second processor coupled to said first processor by means of one of said number of communication channels and coupled to said external device by means of a communication channel, said second processor running a second application in said second memory, said second application adapted to develop said first application, including communicating data to said external device, said data controlling said external device, said data received over said one of said number of communications channels and communicated over said communications channel.

24. A method for analyzing a processor-controlled system, said method comprising:

coupling a first processor to a second processor by means of a first communications channel;

coupling an external device to said second processor by means of a second communications channel, said external device adapted to be controlled by said first processor by means of said first communications channel;

generating data to control said external device;

communicating said data from said first processor to said second processor over said first communications channel;

receiving said data in said second processor and communicating said data via said second communications channel to said external device; and

sending second data to analyze said first processor between said first and second processors via said first communications channel.

25. An apparatus for analyzing a processor-controlled systems, said apparatus comprising:

a plurality of processor-controlled systems interconnected by means of a network;

a registry, coupled to said network, said registry adapted to register processor-controlled systems;

a processor, coupled to said network and registered with said registry, said processor having a user input device, adapted to receive a command on said user input device and adapted to establish a communication channel with a one of said plurality of processor-controlled systems specified in said command.

26. In a target/host computer system, an agent having a size less than 300 kilobytes (Kb).

27. The computer system of claim 26 wherein said agent has a size less than 200 Kb.

28. The computer system of claim 26 wherein said agent has a size less than 100 Kb.

29. The computer system of claim 26 wherein said debugging agent has a size of about 20 Kb.

30. A method for debugging a software program under test, said method comprising the steps of:

executing on a first processor development software having an interface;

loading said software program under test in the memory of a second processor having a receptacle for receiving hardware capable of being read or written;

executing software for examining said software program under test on said second processor;

placing an emulator of said hardware in said receptacle; and

communicating between said first and second processors by means of said emulator.

31. The method of claim 30 wherein said step of communicating comprises

communicating between said first and second processors by means of said interface and said software for examining said software program under test.

32. A system for debugging a software program under test, said system comprising:

a first processor having a memory wherein is located development software having an interface;

a second processor having

a receptacle for receiving hardware capable of being read or written and

a memory wherein is located said software program under test and software for examining said software program under test; and

an emulator of said hardware, located in said receptacle and communicatively coupling said first and second processors.

33. The system of claim 32 wherein said emulator comprises

an emulator communicatively coupling said development software and first and said software for examining.

BEST AVAILABLE COPY

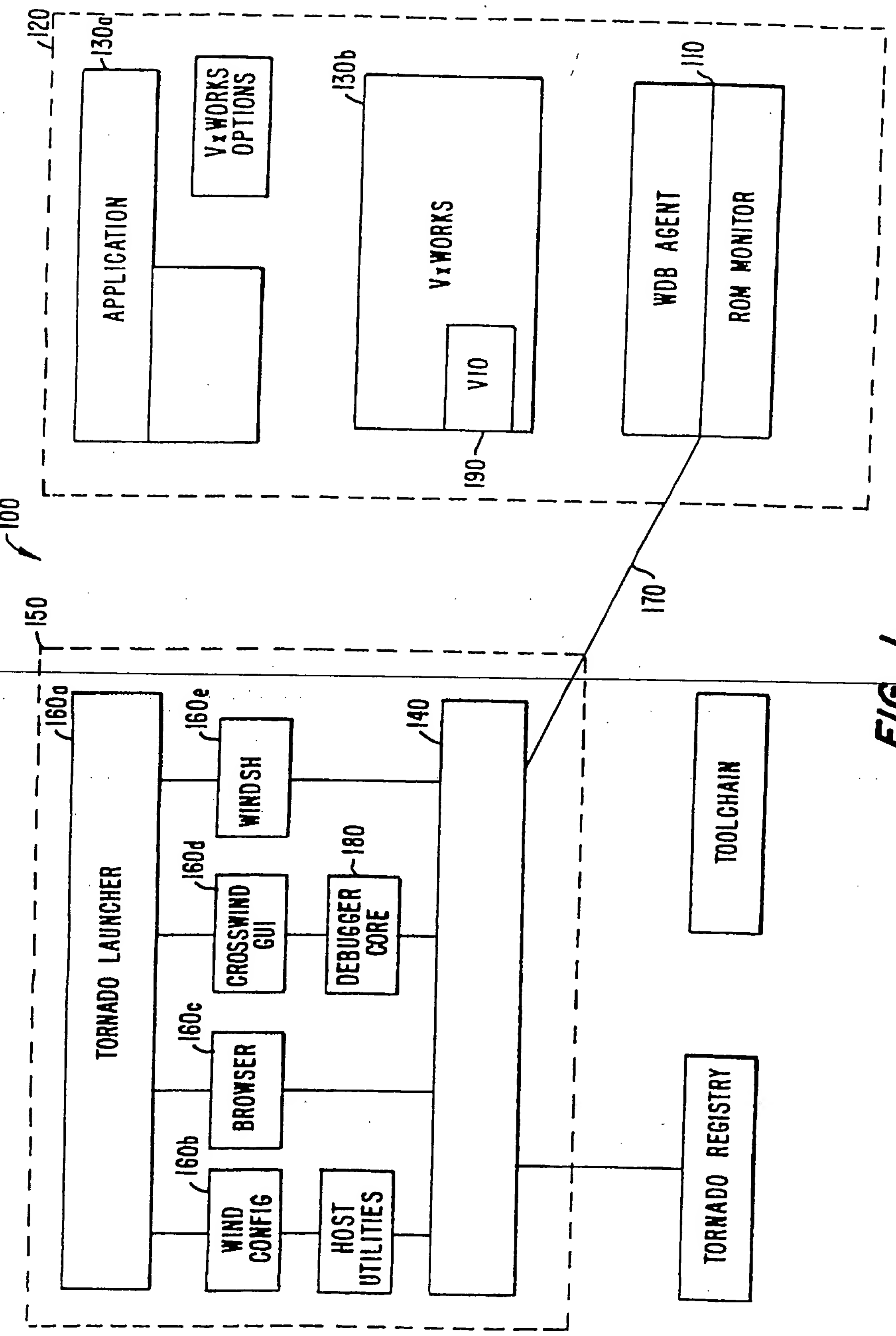
120
130a
APPLICATION
VxWORKS OPTIONS
130b
VxWORKS
VIO
190
110
WDB AGENT
ROM MONITOR
100
170
150
160a
TORNADO LAUNCHER
160e
WINDSH
160d
CROSSWIND GUI
180
DEBUGGER CORE
160c
BROWSER
160b
WIND CONFIG
HOST UTILITIES
140
TOOLCHAIN
TORNADO REGISTRY

FIG. 1.

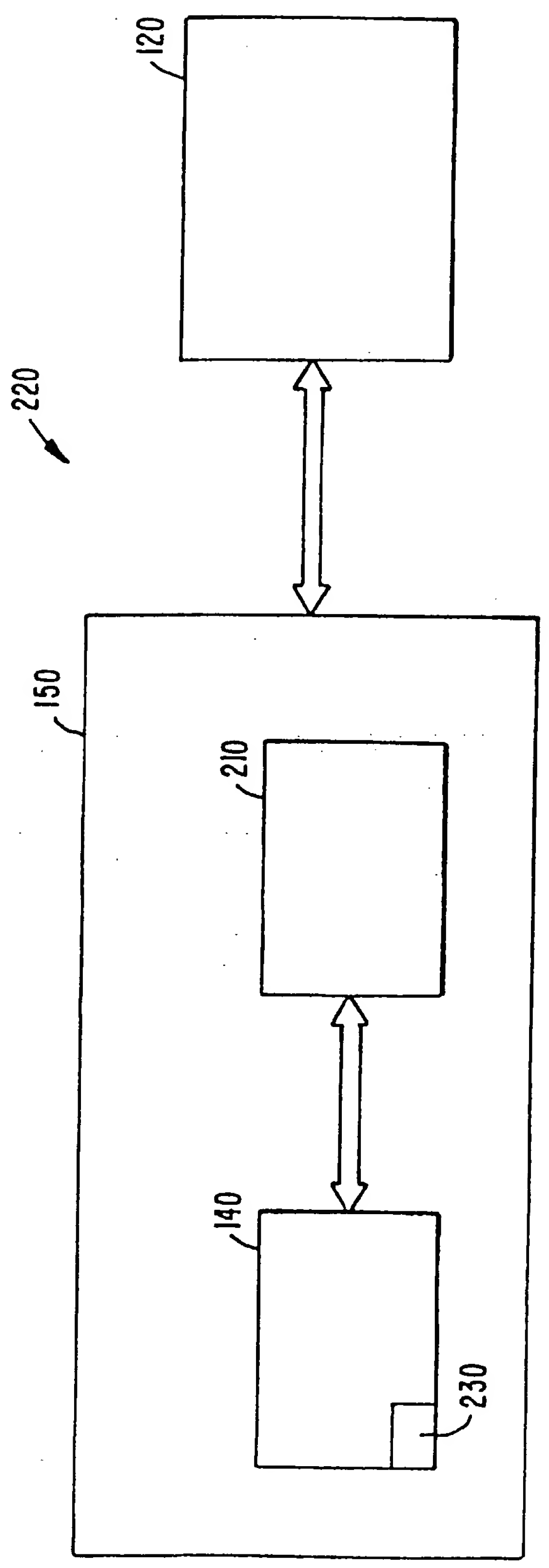

FIG. 2.

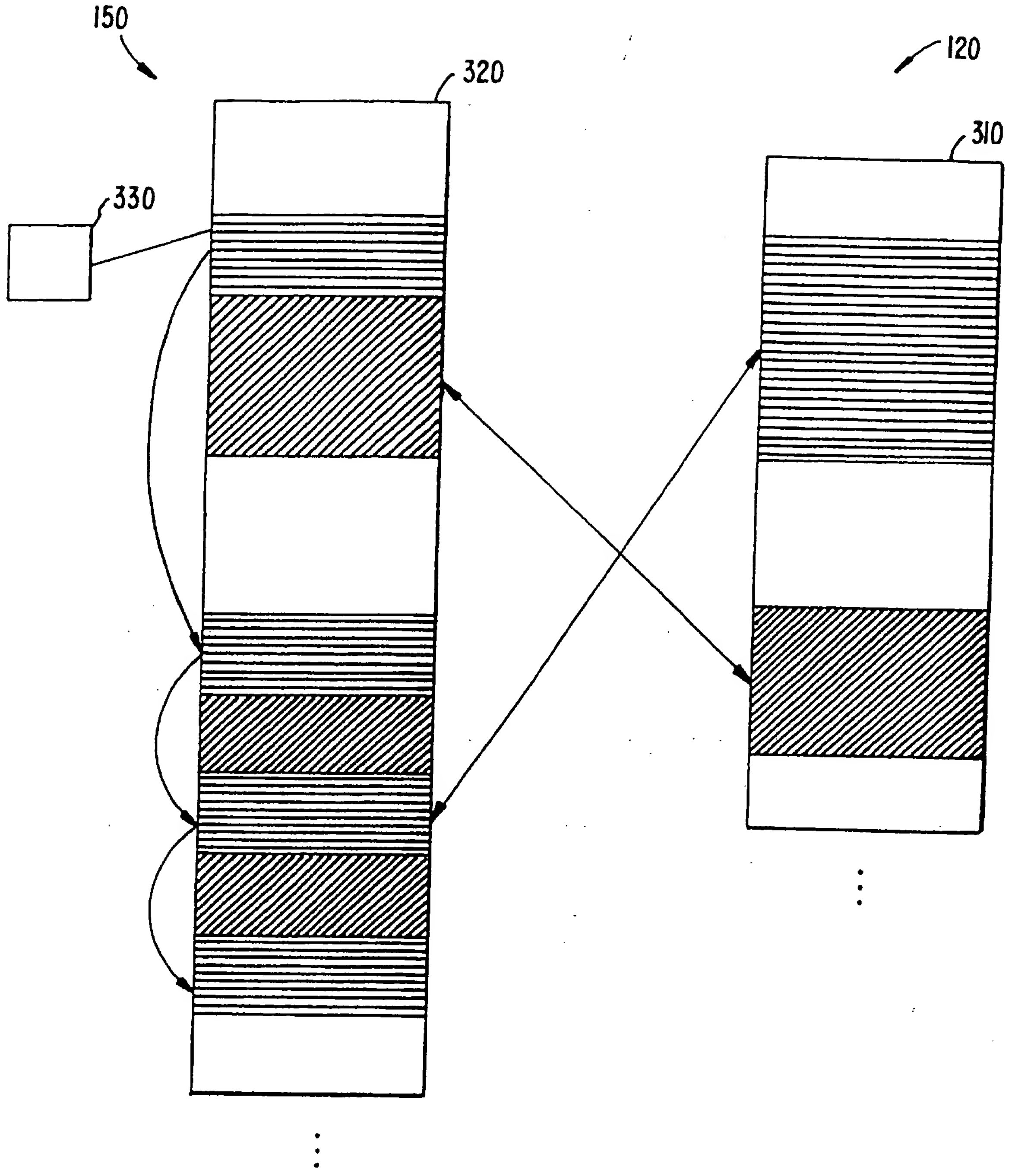

FIG. 3.

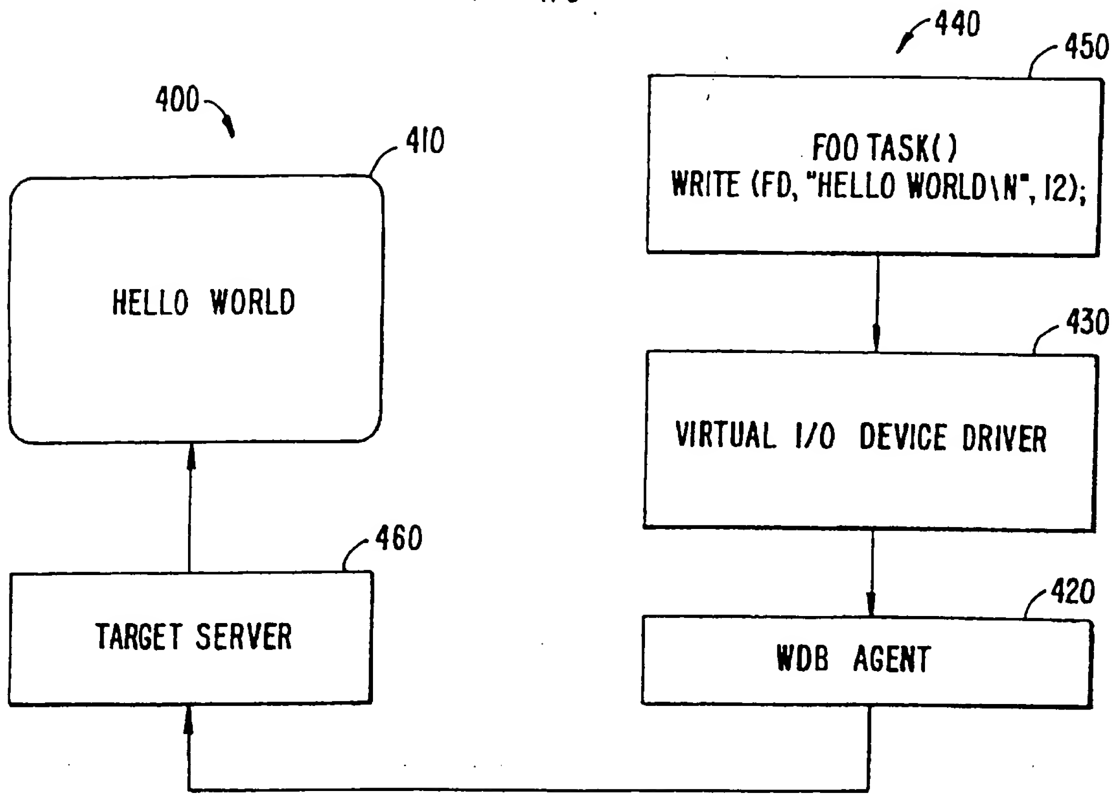

FIG. 4.

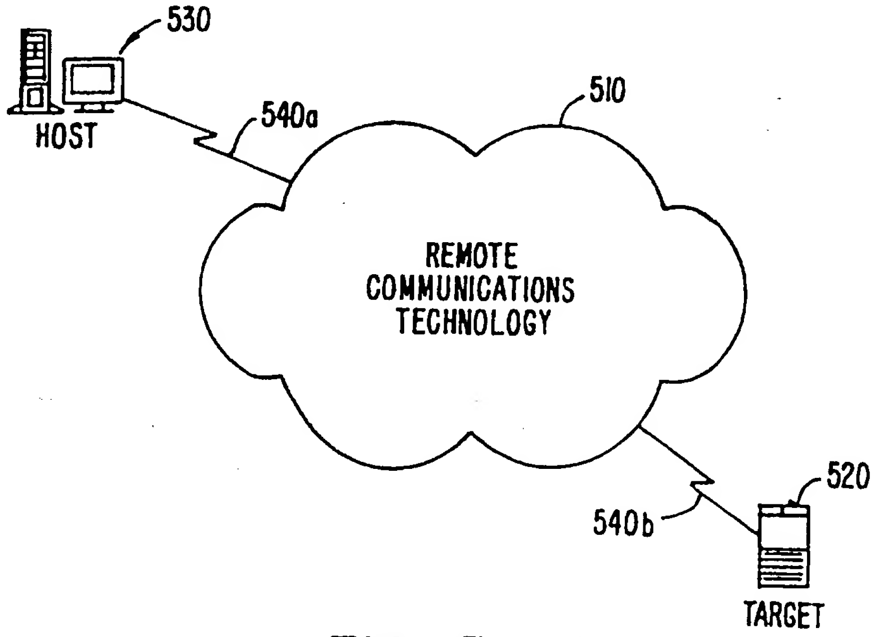

FIG. 5.

SUBSTITUTE SHEET (RULE 26)

BEST AVAILABLE COPY

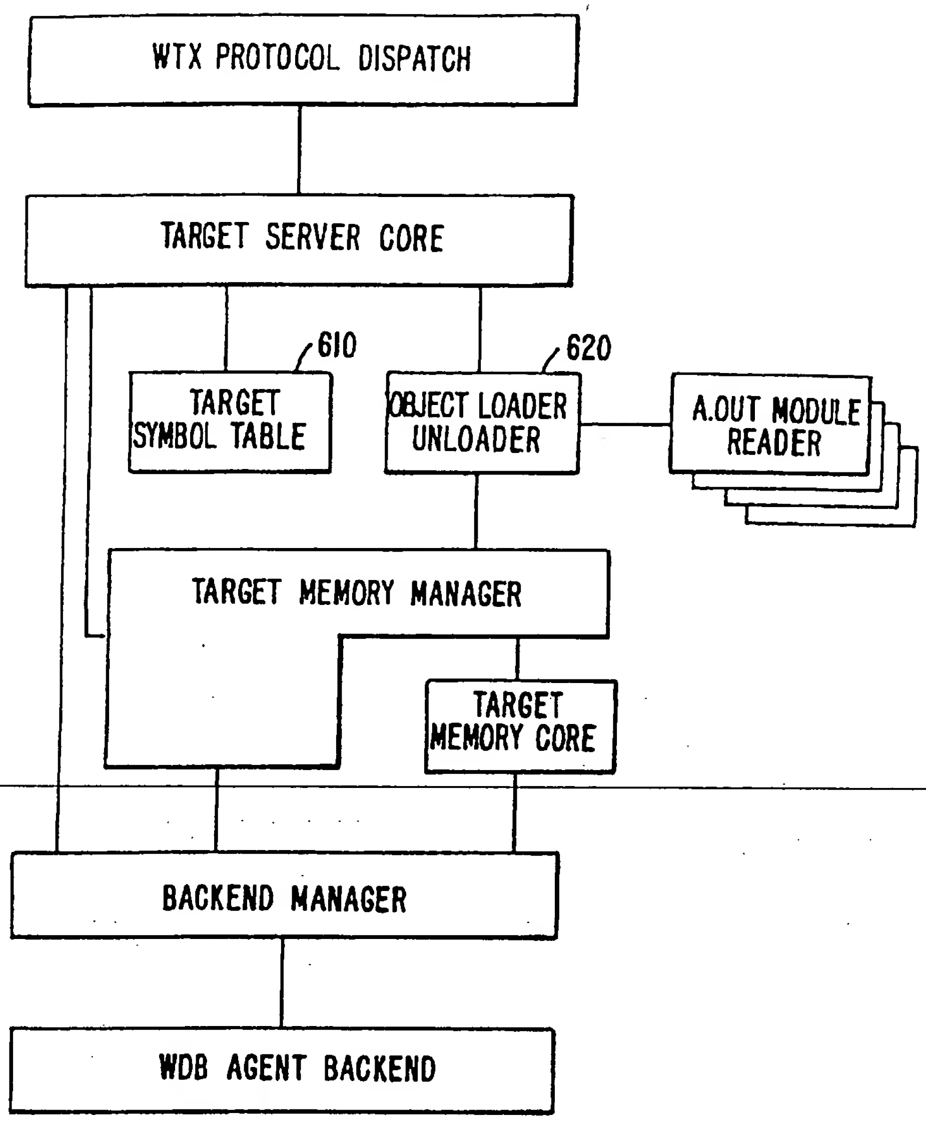
WTX PROTOCOL DISPATCH
TARGET SERVER CORE
610
TARGET SYMBOL TABLE
620
OBJECT LOADER UNLOADER
A.OUT MODULE READER
TARGET MEMORY MANAGER
TARGET MEMORY CORE
BACKEND MANAGER
WDB AGENT BACKEND

FIG. 6.